Parveen Khurana

124 Followers About Following

Activation Functions and Initialization Methods

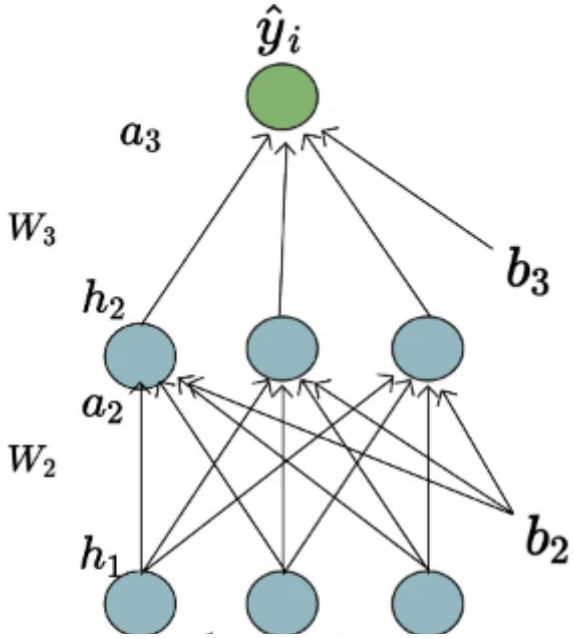
P Parveen Khurana Feb 4, 2020 · 15 min read

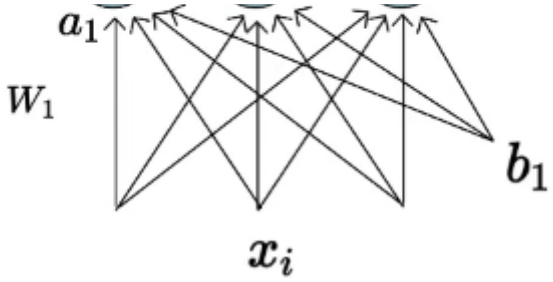
This article covers the content discussed in the Activation Functions and Initialization Methods module of the Deep Learning course and all the images are taken from the same module.

In this article, we discuss a drawback of the Logistic function and see how some of the other choices for the Activation function deal with this drawback and then we also look at some of the common ways to initialize the parameters(weights) of the network which then helps in the training.

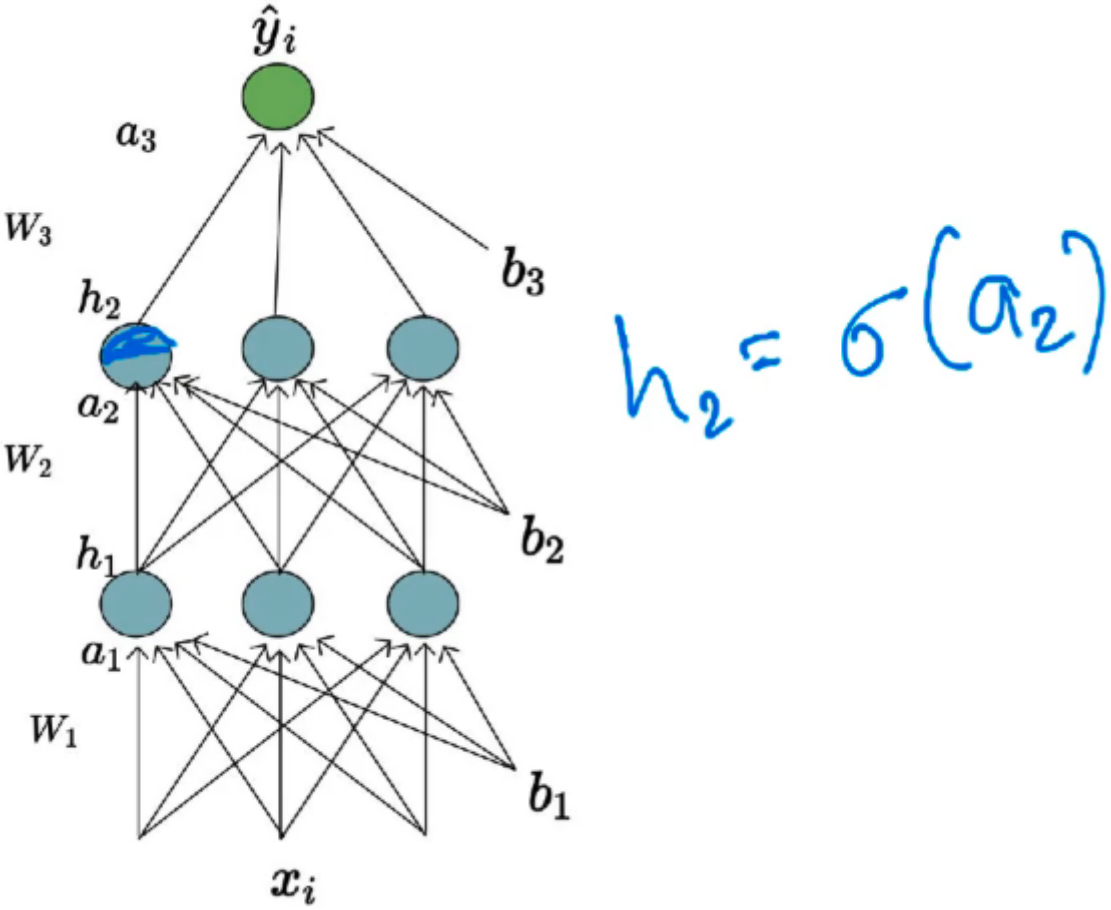
Why are activation functions important?

Let's say we have the below network architecture:





In this network, we compute **'h2'** as:



i.e we apply sigmoid over **'a2'** where **'a2'** is the weighted aggregate of the inputs from the previous layer(first intermediate layer). This sigmoid function is a non-linear function.

What happens if there are no non-linear activation functions in the network ?

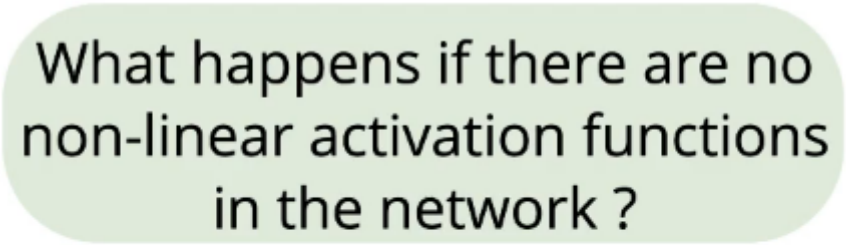
So, we can take a simple scenario where we have **'h2'** is equal to **'a2'** or in other words, we are passing **'a2'** as it is to the neuron in the output layer and we are not applying any non-linearity on **'a2'**.

As we are not applying any non-linearity then our final output looks like:

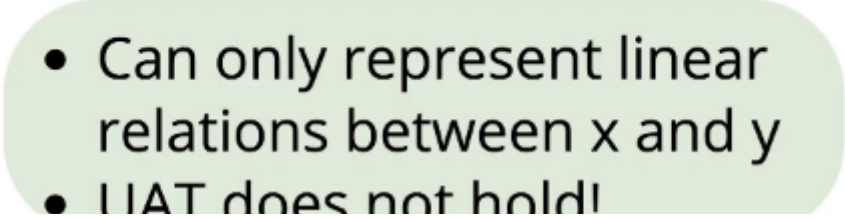
$$\hat{y}_i = W_3(W_2(W_1 x_i)) = W x_i$$

We are multiplying the input with one number 'W_1', then by another number 'W_2' and then by 'W_3' all of which are just matrices and we can write their product as '**W**'.

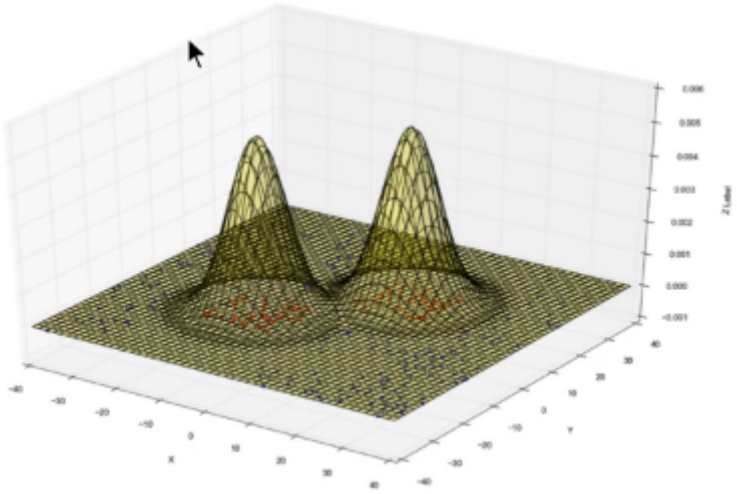
And **the output is just a linear transformation of the input**. So, the family of functions that we are learning in this scenario(when we are not applying non-linearity at any level) even though it's a very very Deep Neural Network is just a linear function or family of linear functions.



$$\hat{y}_i = W_3(W_2(W_1 x_i)) = W x_i$$



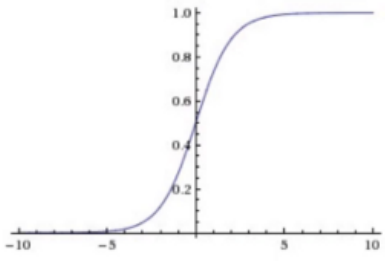
- The representation power of a deep NN is due to its non-linear activation functions!

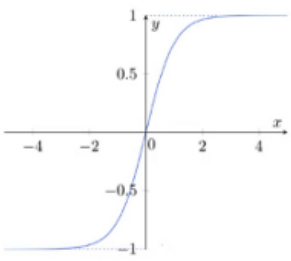


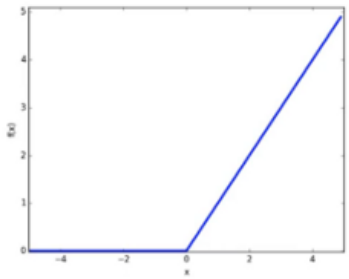
If non-linear activation function is not there then our model can not deal with linearly inseparable data or say that our model cannot learn a function where the relationship between input and output is a non-linear function. So, this is the reason why activation functions are important and we need non-linear activation functions in our model.

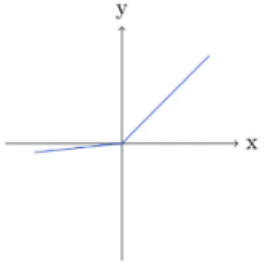
Some of the commonly used activation functions are plotted below:

- logistic
- tanh
- ReLU
- Leaky ReLU



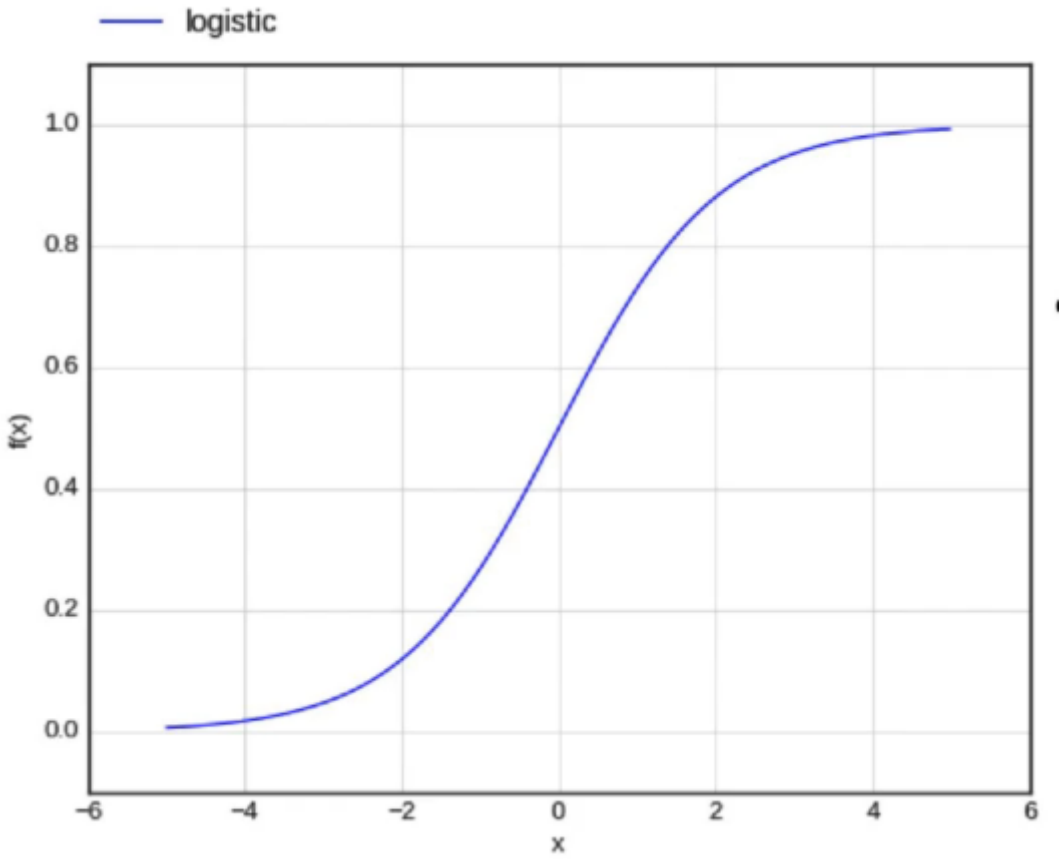






Saturation in Logistic Neuron:

The logistic function looks like:



And the function equation is:

$$f(x) = \frac{1}{1+e^{-x}}$$

that means if we have a neuron in DNN and we want to compute the activation value from pre-activation(pre-activation itself would be a linear combination of the inputs) value, then we need to apply the below function:

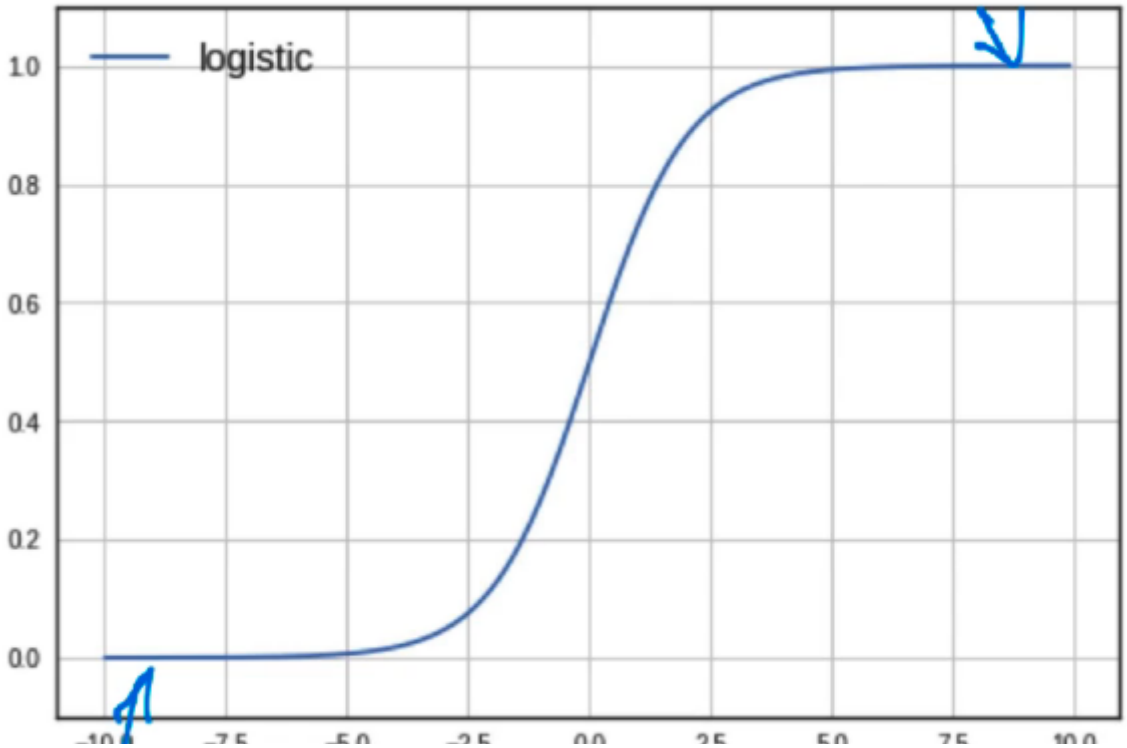
If **'x'** is the input to the logistic function then the derivative of the logistic function with respect to **'x'** is given by:

$$f(x) = \frac{1}{1+e^{-x}}$$

$$f'(x) = \frac{\partial f(x)}{\partial x} = f(x) * (1 - f(x))$$

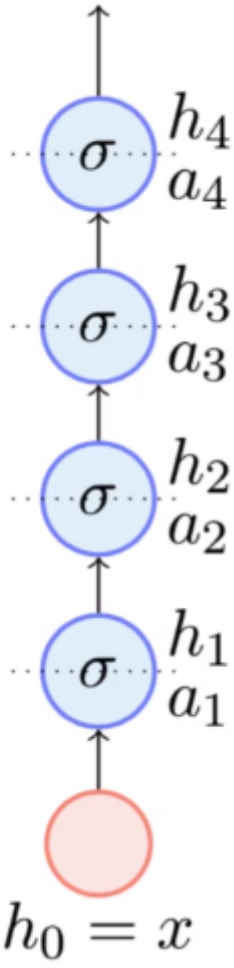
The derivative is going to be used in the Gradient Descent update rule so we must know what the derivative value for each of the activation looks like.

A neuron is said to be saturated when it reaches its peak value, so if we plug in a large value in the logistic function, then the function would almost reach its peak value i.e 1. Similarly, if we plug in a large negative value, then again the function would reach its peak value 0. And **in either of the cases when logistic function reaches its peak value, if we plug in the values into the equation for the derivative of the function, we find that the derivative value would be 0.**



$$\textit{Saturation: when } f(x) = 0 \textit{ or } 1$$
$$\textit{and hence } f'(x) = 0$$

Let's understand the implication of this using a simple neural network:

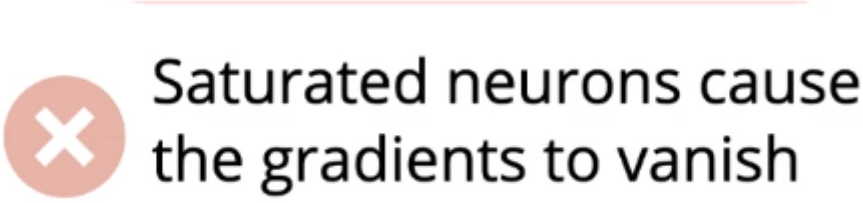


$$a_3 = w_2 h_2$$
$$h_3 = \sigma(a_3)$$

Now, let's say we want to compute the derivative of the Loss function with respect to **'a3'**. So, we can compute this derivative using chain rule i.e first we compute the derivative of the loss function with respect to **'h4'** multiply it with the derivative of **'h4'** with respect to **'a4'** multiply it with the derivative of **'a4'** with respect to **'h3'** then by derivative of **'h3'** with respect to **'a3'**.

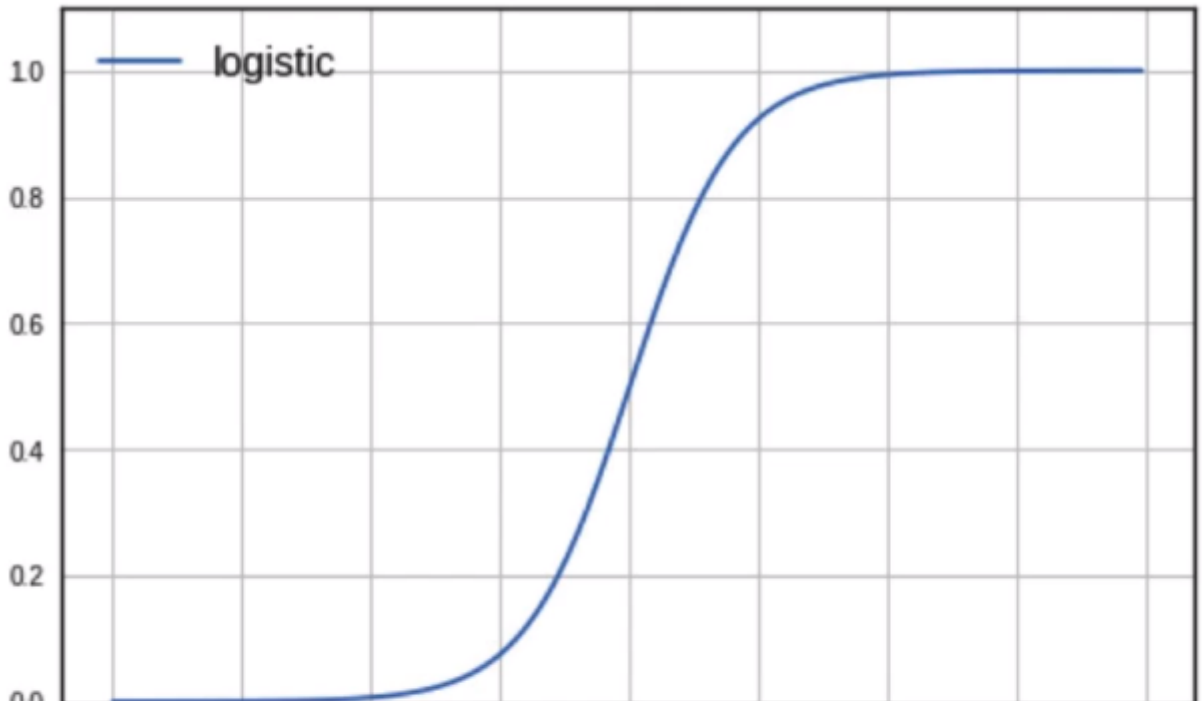
The issue is if the neuron has saturated than the **'h2'** would be close to 0 or 1 and the derivative would then be 0 which means the entire derivative value would be 0 and since the update equation of the parameter is directly proportional to the gradient/derivative; this implies that parameters would not get updated if the neuron has saturated.

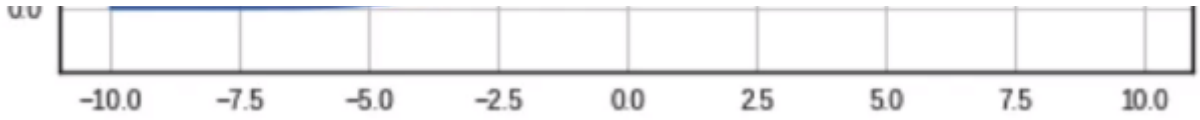
So, **if the neuron has saturated, weights would not get updated. This is known as the Vanishing Gradient problem.**



Let's see why the neuron would saturate in the first place:

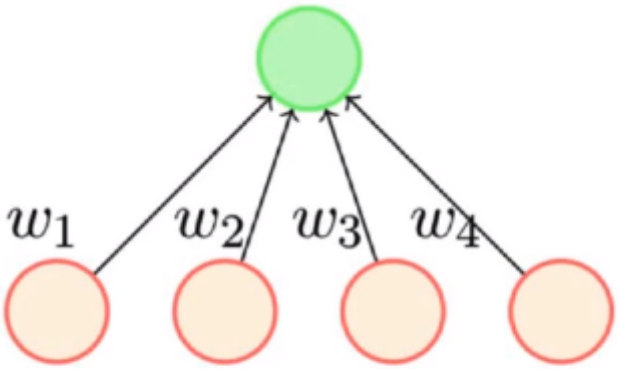
If we plot the activation values on the x-axis and the y-axis corresponds to logistic function applied over activation value, we would get the following plot:





$$a = \sum_{i=1}^{4} w_i x_i$$

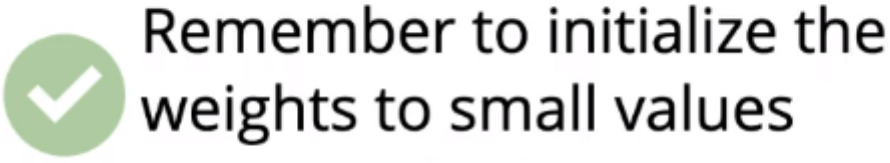
This activation is the weighted sum of all the inputs.



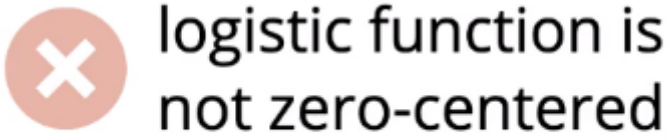
$$\sigma\left(\sum_{i=1}^{4} w_i x_i\right)$$

xi's(inputs) **are going to be normalized meaning they would be standardized values between 0 to 1 so they are not going to blow up.** Now, suppose we happen to initialize the weights to very large quantities and if we have 100 such weights corresponding to 100 inputs, in that case, the activation value would blow up say even if it reaches 10 or 11, then as per the above plot(and as per logistic function equation), the neuron is going to saturate. And similarly, if we initialize the weights to very large negative values, in that case, the activation value would be a large negative value and the neuron would saturate in this case as well.

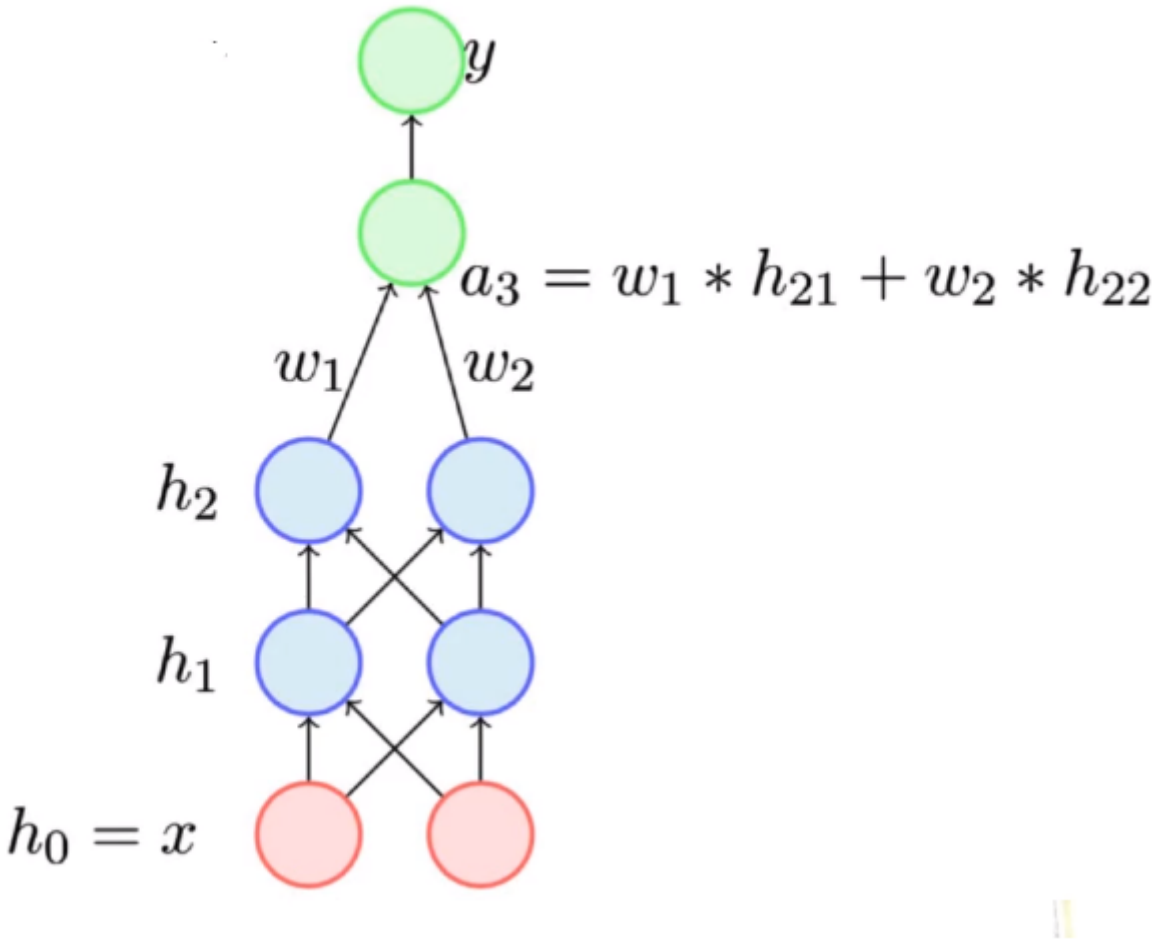
So, **if the weights are very large or very small(negative) then the activation would be very large or small and the neuron would saturate either on the positive side or in the negative direction.**



Apart from saturation, other problems with logistic functions are:



Logistic function only takes on positive values between 0 and 1. **The zero-centered function would take on positive value for some inputs and it would take on negative values for some inputs.** Let's see the problem associated with a function not being zero-centered by taking a simple neural network:



We are looking at **'a3'** value which is the weighted sum of the output from the previous layer. And now we want to compute the derivative of the Loss function with each of the weights **'w1'** and **'w2'**. The derivative as per the chain rule would look like:

$$\nabla w_1 = \frac{\partial \mathscr{L}(\mathbf{w})}{\partial y} \frac{\partial y}{h_3} \frac{\partial h_3}{\partial a_3} \frac{\partial a_3}{\partial w_1}$$

$$\nabla w_2 = \frac{\partial \mathscr{L}(\mathbf{w})}{\partial y} \frac{\partial y}{h_3} \frac{\partial h_3}{\partial a_3} \frac{\partial a_3}{\partial w_2}$$

We can see that the some part of the derivative(in red in the above image) is common for both the derivatives.

Now **'a3'** is given by the below equation

$$a_3 = w_1 * h_{21} + w_2 * h_{22}$$

And its derivative with respect to **'w1'** and **'w2'** would be just **'h21'** and **'h22'**

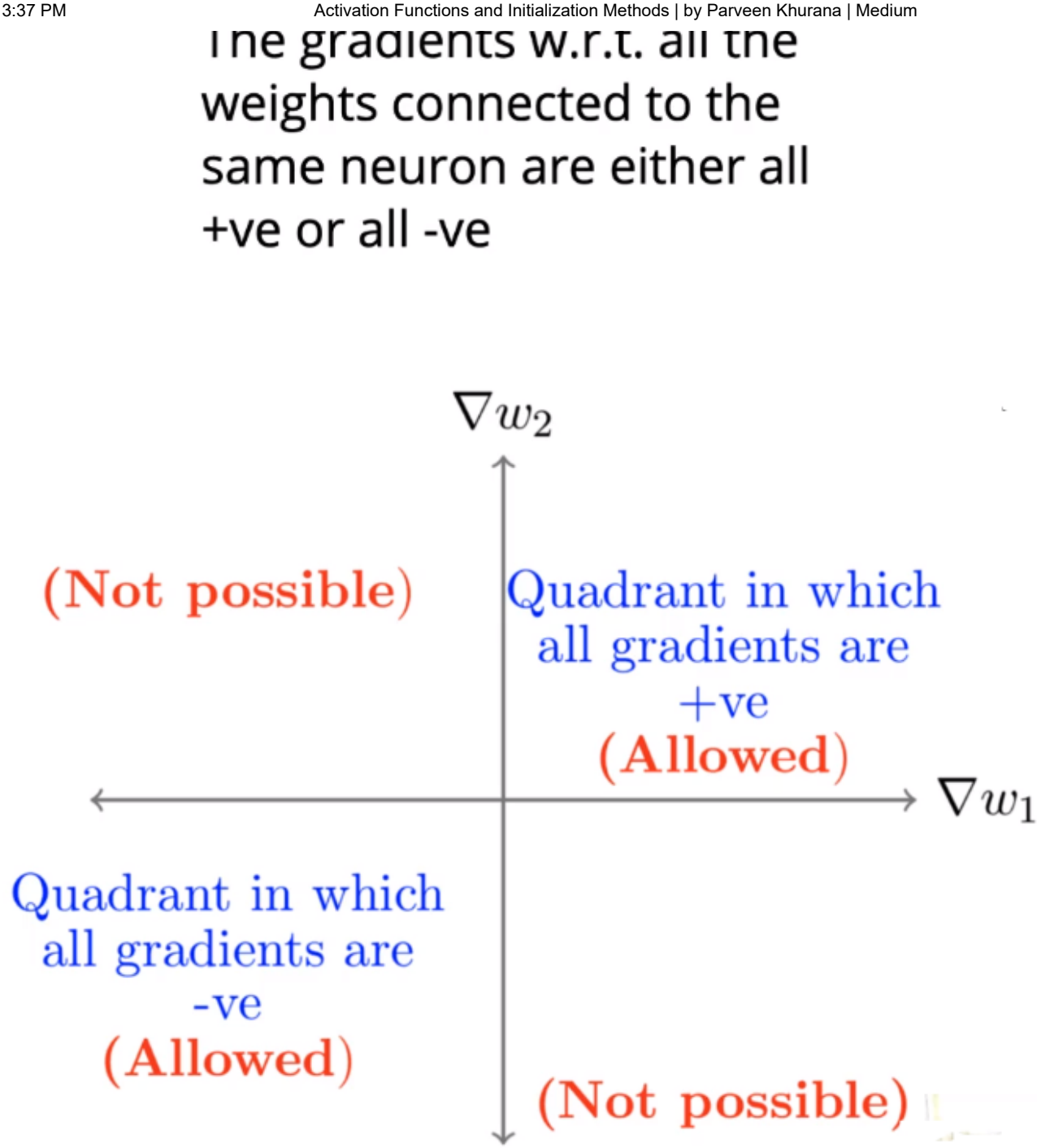
$$\nabla w_1 = \frac{\partial \mathscr{L}(\mathbf{w})}{\partial y} \frac{\partial y}{h_3} \frac{\partial h_3}{\partial a_3} \; h_{21}$$

$$\nabla w_2 = \frac{\partial \mathscr{L}(\mathbf{w})}{\partial y} \frac{\partial y}{h_3} \frac{\partial h_3}{\partial a_3} \; h_{22}$$

Now **'h21'** and **'h22'** are the outputs of the logistic function, which means both of them are going to be between 0 to 1 meaning both of these quantities are going to be positive. So, in the above image, red terms are common for the derivatives with respect to both the weights whereas the terms in blue can only be positive, they can not be negative.

Now suppose if any of the terms in the red quantity is negative that means the entire value of the derivative/gradient would be negative for both the weights and if all the terms in red are positive and then again both the derivatives/gradients are going to be positive. So, **we have two possibilities either both are derivatives(derivatives with respect to both the weights) are positive or both the derivatives are negative**.

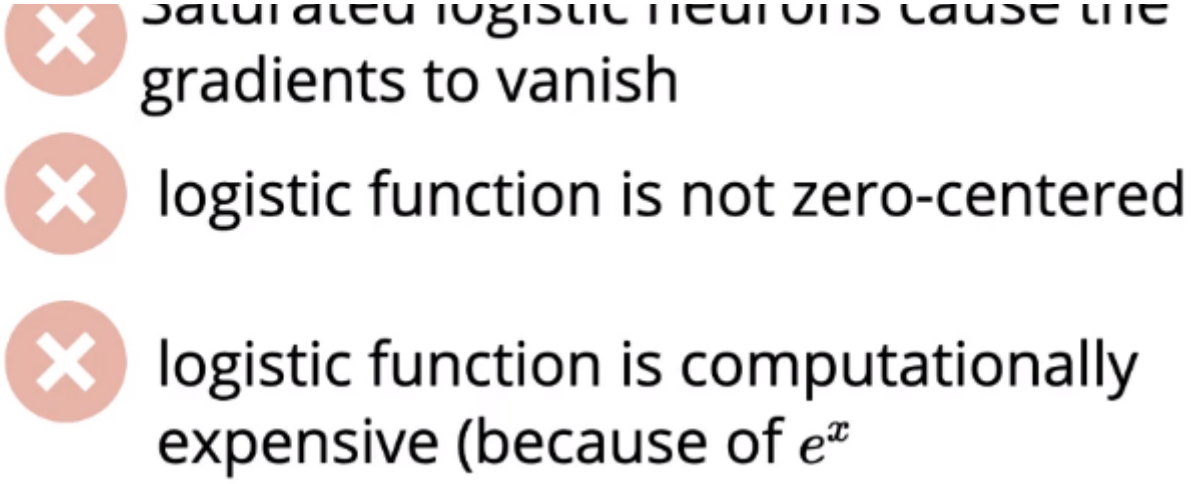
The gradients w.r.t. all the



The overall gradient vector(which would contain the gradient with respect to both the weights) could lie either in the first quadrant or in the third quadrant. What this means is that in the update equation for the theta, there is the restriction on the possible directions that we can have, in particular, we can only have the direction either in the 1st or 3rd quadrant. So, there is a certain restriction during the training, we could only move in a certain direction, we can not move in all possible directions that we would like to move to faster reach convergence.

So, the problems with the Logistic function are:

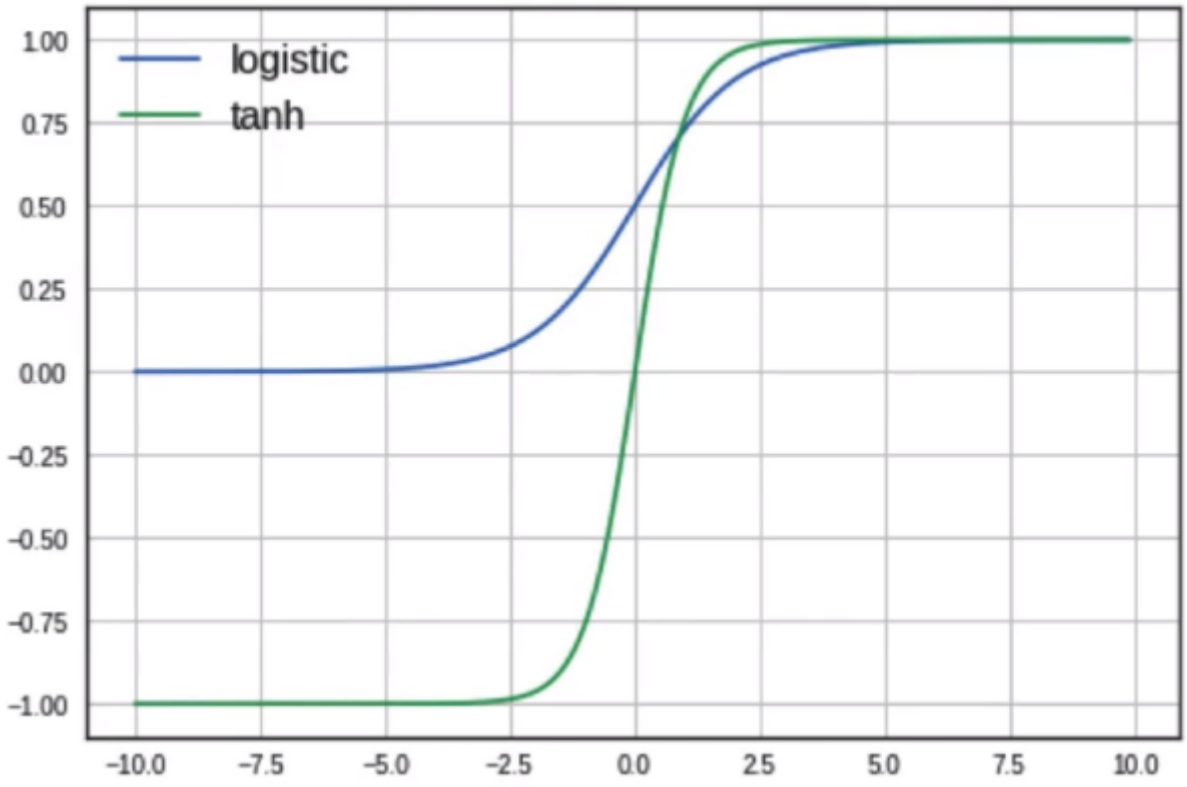
Saturated logistic neurons cause the



Introducing Tanh and ReLu:

The equation of **Tanh** function is:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



The green curve in the above plot corresponds to **tanh** function. Both the functions in the above plot are S-shaped but their ranges are different; Logistics(**Sigmoid**) ranges

from 0 to 1 whereas **tanh** lies between -1 and 1.

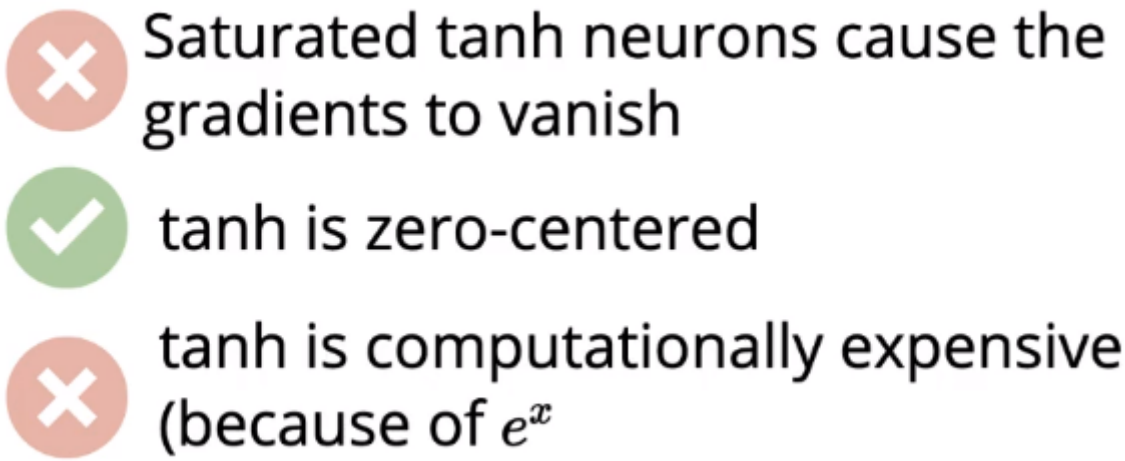
A very large value of '**x**' would give the output as 1 and a small value of '**x**' would give the output as -1 in case of **tanh**.

The derivative of the function(**tanh)** with respect to '**x**' is given as:

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

$$f'(x) = \frac{\partial f(x)}{\partial x} = (1 - (f(x))^{2})$$

The derivative is going to be 0 at the peak values of the function i.e for **f(x)** equal to 1 or -1. So, **tanh** function could also saturate which would result in the vanishing gradients issue.



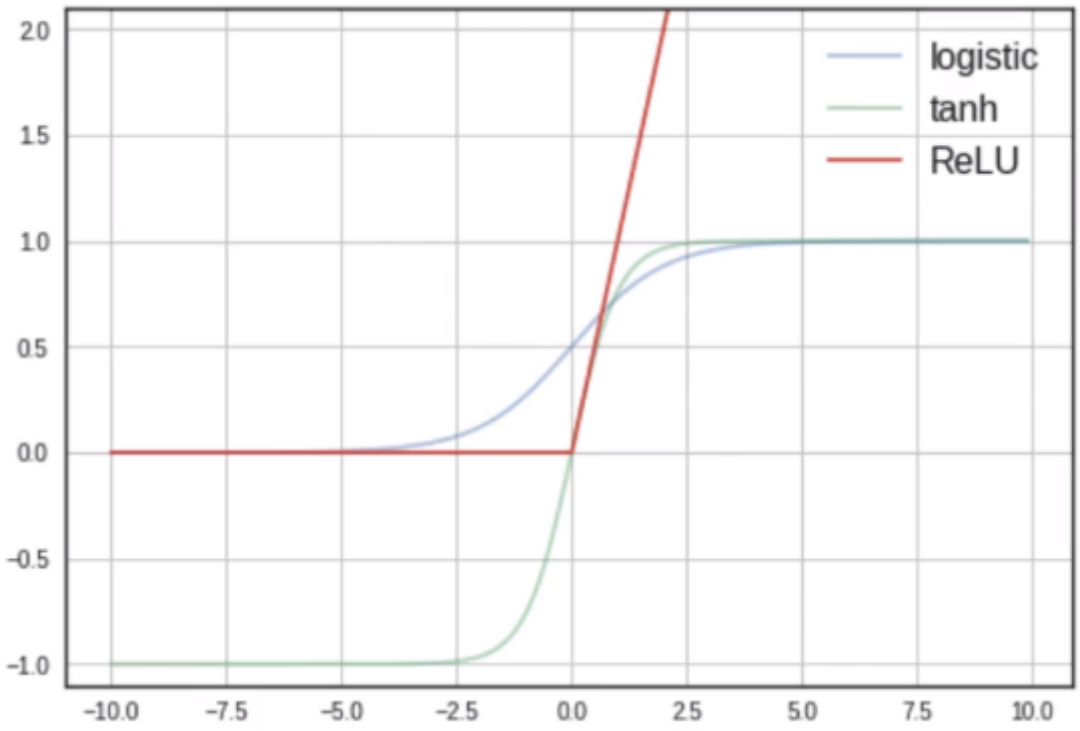
tanh is preferred over logistic function.

ReLU function:

The function equation is given as:

$$f(x) = max(0, x)$$

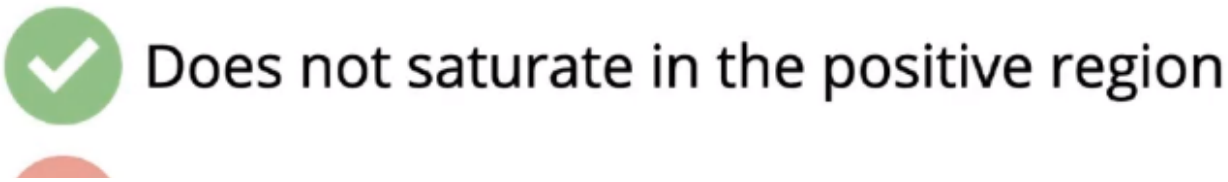
If the input is positive then the ReLU function will output the number itself and if the input is negative then the ReLU function would output 0:

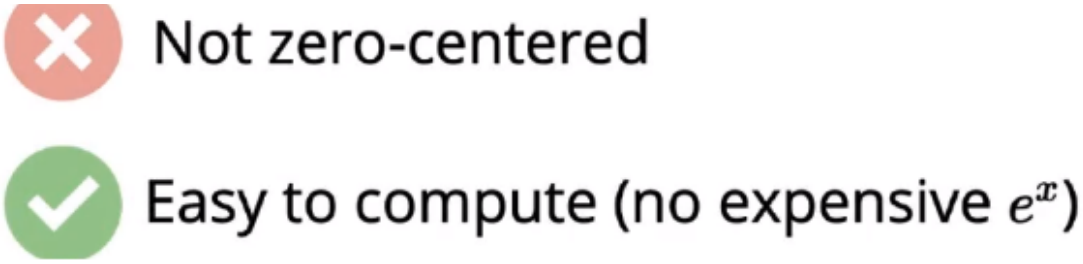


The derivative of this function looks like:

$$f'(x) = \frac{\partial f(x)}{\partial x} = \begin{cases} 0 & if \quad x < 0 \\ 1 & if \quad x > 0 \end{cases}$$

This function does not saturate in the positive region(positive inputs) so there would be no vanishing gradients in the positive region. This function is not equally divided along the x-axis or in other words, this function is not a zero centered function.

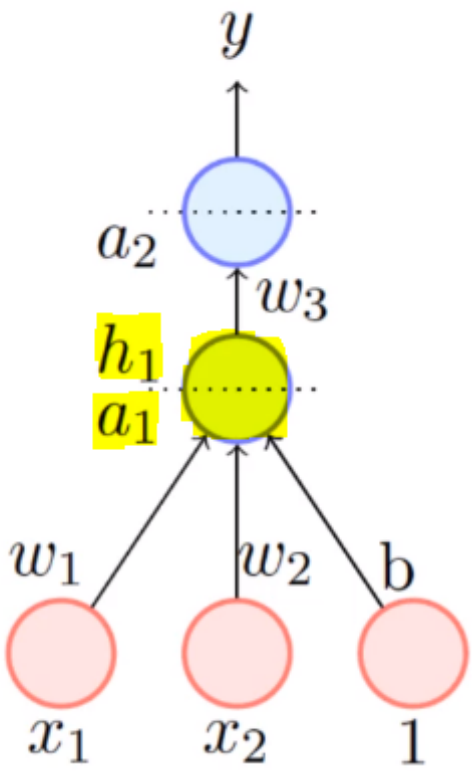




ReLU is the standard activation function to be used with CNN.

A caveat in using ReLU:

Let's start with a simple network as shown below and focus on the yellow highlighted layer/neuron in the below network:



At this layer, we have used the ReLU activation function that means

$$h_1 = ReLU(a_1) = max(0, a_1)$$
$$= max(0, w_1 x_1 + w_2 x_2 + b)$$

What happens if h_1 takes on a large

What happens if b takes on a large negative value due to a large negative update (∇b) at some point ?

$$w_1 x_1 + w_2 x_2 + b < 0 \quad [if \quad b << 0]$$
$$\implies h_1 = 0 \quad [dead\ neuron]$$
$$\implies \frac{\partial h_1}{\partial a_1} = 0$$

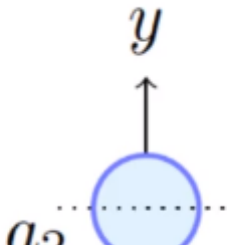
If this derivative of **'h1'** with respect to **'a1'** is 0, then the value of the gradient of the loss function with respect to **'a1'** would be 0 as per the below equation:

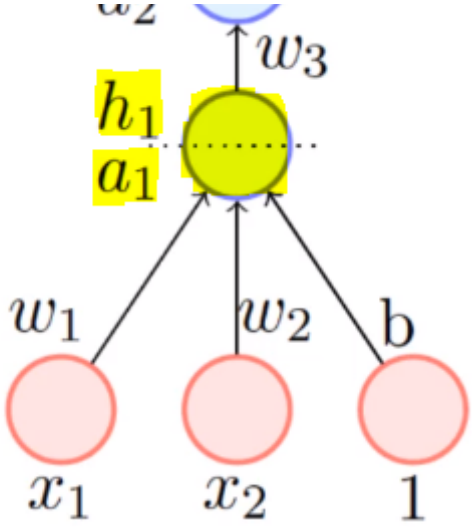
$$\nabla w_1 = \frac{\partial \mathscr{L}(\theta)}{\partial y} \cdot \frac{\partial y}{\partial a_2} \cdot \frac{\partial a_2}{\partial h_1} \cdot \frac{\partial h_1}{\partial a_1} \cdot \frac{\partial a_1}{\partial w_1}$$

And now as per the update rule for **'w1'**, **'w1'** would not change if the gradient with respect to **'w1'** is going to be 0

$$w_1 = w_1 - \eta \nabla w_1$$

Similarly the derivative of the loss function with respect to **'w2'**





would have the term of the derivative with respect to **'a1'** which is 0 so the gradient with respect to **'w2'** would also be 0.

$$\frac{\partial h_1}{\partial a_1}$$

And the same thing would happen for the derivate with respect to **'b'**. So, in effect, none of the parameters **'w1'**, **'w2'**, **'b'** would get updated. That means **'b'** would continue to be a large negative value and since the inputs **'x1'** and **'x2'** are standardized that means they are always going to lie between 0 and 1 and they would not be able to compensate for a large negative value of **'b'** and there is a great probability that the quantity of left side in the below equation would always remain to be less than 0.

$$w_1 x_1 + w_2 x_2 + b < 0$$

So, once a neuron becomes dead, then the weights associated with that would not get updated. If we pass a new input through this neuron, once again the activation value(equation in the above image) would be less than 0, again the neuron would be dead and weights associated with this neuron would not get updated. **So, once a ReLU neuron gets dead it stays dead throughout the training.** So, that's a caveat in using the ReLU function and this happens because, in the negative region, ReLU would saturate.

$$w_1 x_1 + w_2 x_2 + b < 0 \quad [if \quad b << 0]$$
$$\implies h_1 = 0 \quad [dead\ neuron]$$
$$\implies \frac{\partial h_1}{\partial a_1} = 0$$
$$\implies w_1, w_2, b\ remain\ unchanged$$
$$\implies the\ neuron\ stays\ dead\ forever$$

- A large fraction of ReLU units can die if the learning rate is set too high

- It is advised to initialize the bias to a positive value

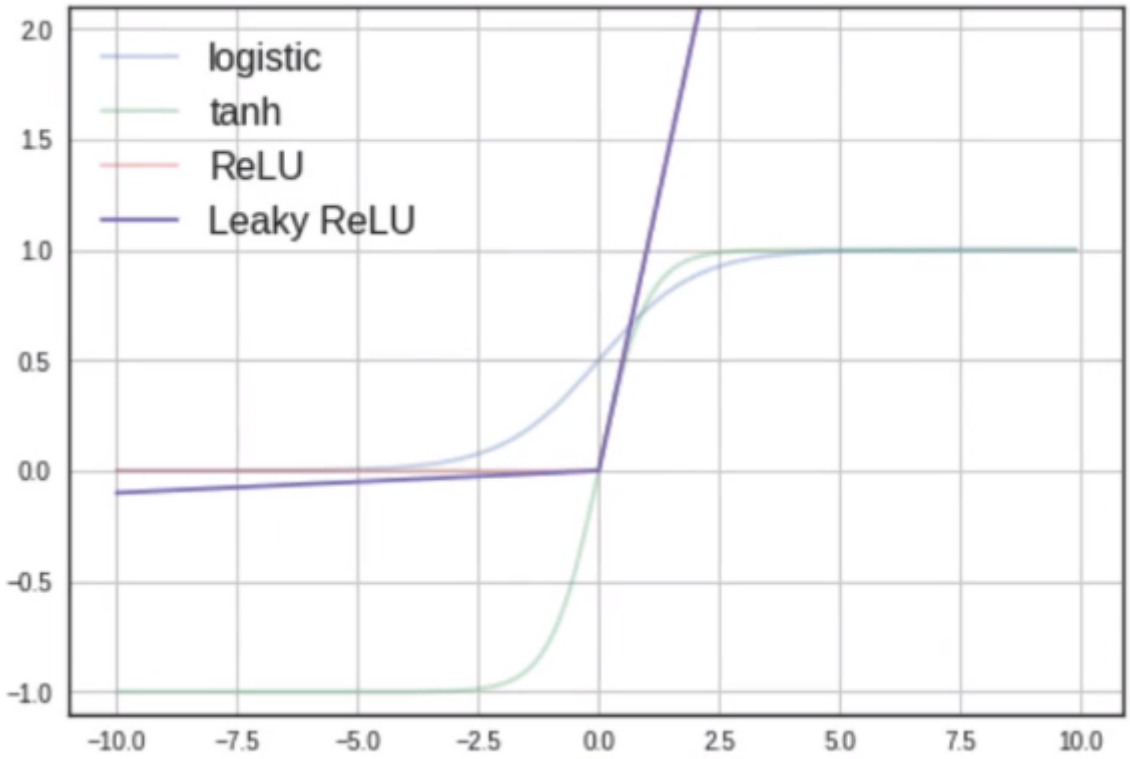
- Use other variants of ReLU

Leaky ReLU:

The function's equation and its derivative with respect to **'x'** are given as:

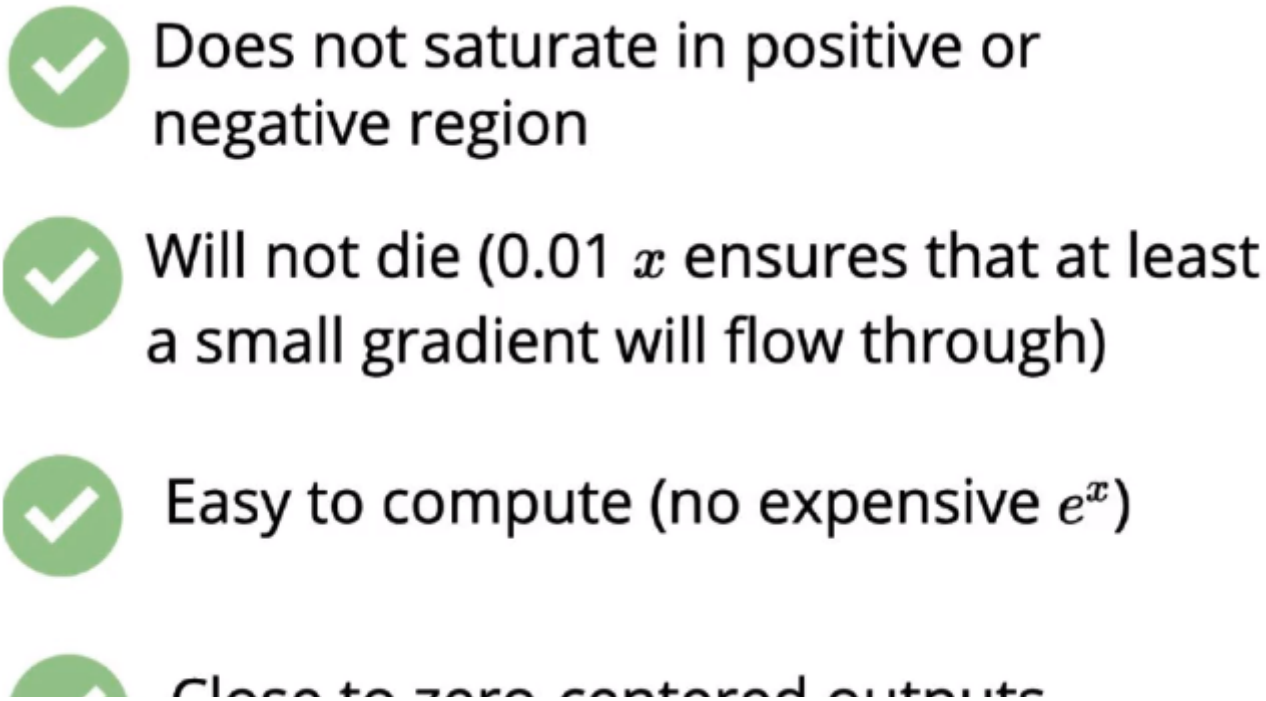
$$f(x) = max(0.01x, x)$$
$$f'(x) = \frac{\partial f(x)}{\partial x} = \begin{cases} 0.01 & if \quad x < 0 \\ 1 & if \quad x > 0 \end{cases}$$

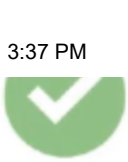
So, in the positive region, **f(x)** is going to be equal to **'x'** whereas, in the negative region, **f(x)** is going to be equal to **'0.01x':**



Now the derivative would be not zero in the negative region, there would be this value of 0.01 in the negative region and after a few iterations, parameters would get out of a large negative value and hence the total sum in the below equation would be greater than 0.

$$w_1 x_1 + w_2 x_2 + b < 0$$



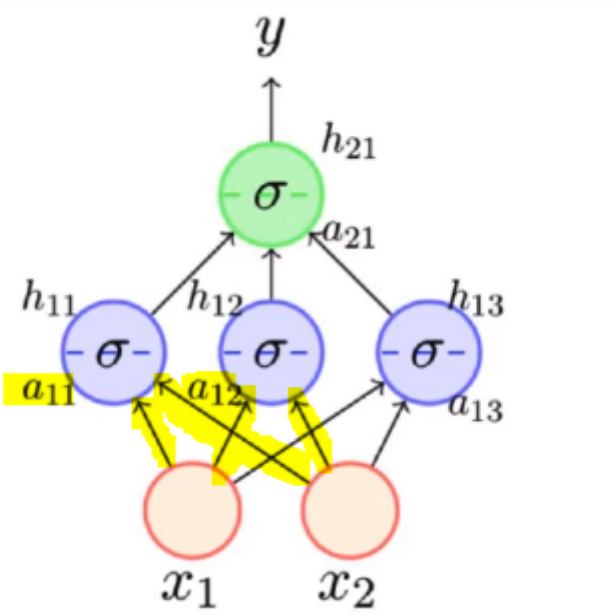


Close to zero-centered outputs

Symmetry Breaking Problem:

Now the question we will try to answer is how to initialize the weights and why not initialize all the weights to 0 as we are just randomly initializing the weights.

Once again we will consider a simple neural network as depicted below:



And we will focus on **a11** and **a12** which have certain weights connected to them and the inputs are connected via these weights to these neurons. Now the values of **a11** and **a12** would be given as(we are not taking bias into account for the sake of simplicity):

$$a_{11} = w_{11}x_1 + w_{12}x_2$$
$$a_{12} = w_{21}x_1 + w_{22}x_2$$

If all the weights are initialized to 0, then **a11** and **a12** both would be 0 and **h11** and **h12** would be equal to each other:

$$a_{11} = a_{12} = 0$$
$$h_{11} = h_{12}$$

Let's see what would be the consequence of this:

We have initialized the weights to 0, passed the inputs through all the layers and computed the predicted output using which along with the true output, we can compute the loss and using loss we can have the value of the gradients to be used in the update equation:

$$\nabla w_{11} = \frac{\partial \mathscr{L}(\mathbf{w})}{\partial y} . \frac{\partial y}{\partial h_{11}} . \frac{\partial h_{11}}{\partial a_{11}} . x_1$$

$$\nabla w_{21} = \frac{\partial \mathscr{L}(\mathbf{w})}{\partial y} . \frac{\partial y}{\partial h_{12}} . \frac{\partial h_{12}}{\partial a_{12}} . x_1$$

$$but \quad h_{11} = h_{12}$$

$$and \quad a_{11} = a_{12}$$

$$\therefore \nabla w_{11} = \nabla w_{21}$$

So, we started with equal values for the weights **w11** and **w21**, then we got equal values for **h11** and **h12**. Hence, the derivatives are also equal which are the derivatives with respect to **w11** and **w21**. And now if we update the weights, they would get updated with the same values and since they were initialized with the same value that means even after the update, they would have the same value.

Now in the next iteration also, **w11** would be equal to **w21** and **w12** would be equal to **w22**, the net effect would be that the **a11** is equal to **a12**, **h11** is equal to **h12** and once again the derivatives/gradients value are going to be equal and the weights would get updated by the same amount.

So, what will happen if we initialize the weights to the same amount, to begin with, is that they will continue to remain equal throughout the training. This is known as the **symmetry-breaking problem**. Once we start with equal values, the weights would continue to remain equal.

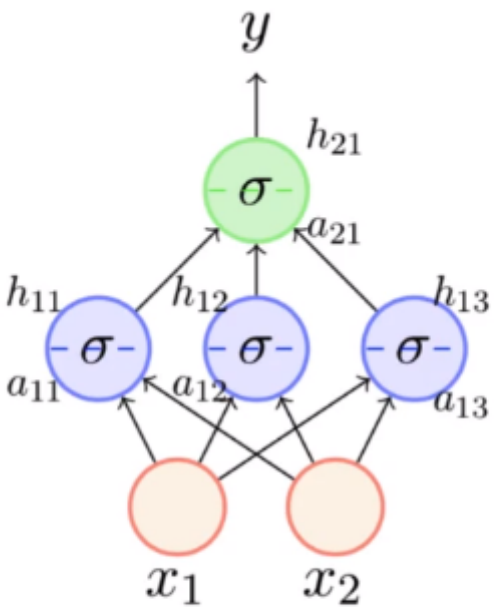
This symmetry will never break during training (**symmetry breaking problem**)

Hence **weights** connected to the same neuron should **never be initialized to the same value**

never initialize all weights to 0

never initialize all weights to same value

Let's see what would be the case when we initialize all the weights to some large values:



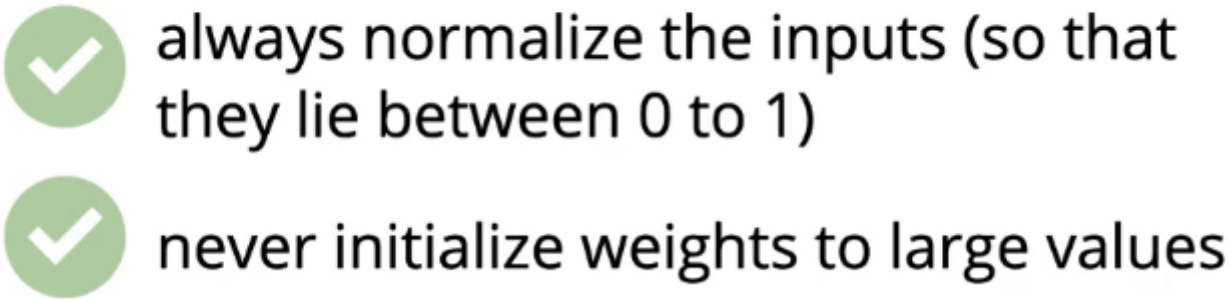
$$a_{11} = w_{11}x_1 + w_{12}x_2$$

$$a_{12} = w_{21}x_1 + w_{22}x_2$$

The inputs **x1** and **x2** would be normalized and would lie between 0 and 1.

Now if the weights are large, then still the weighted sum of inputs(pre-activation) could take on a large value especially if we have a large input neurons and we know this is going to be an issue at least in the case of **logistic** and **tanh** neurons because if the weighted sum blows up, then the neuron would saturate which will result in vanishing gradient problem. So, that's the problem with the weights being initialized to large values.

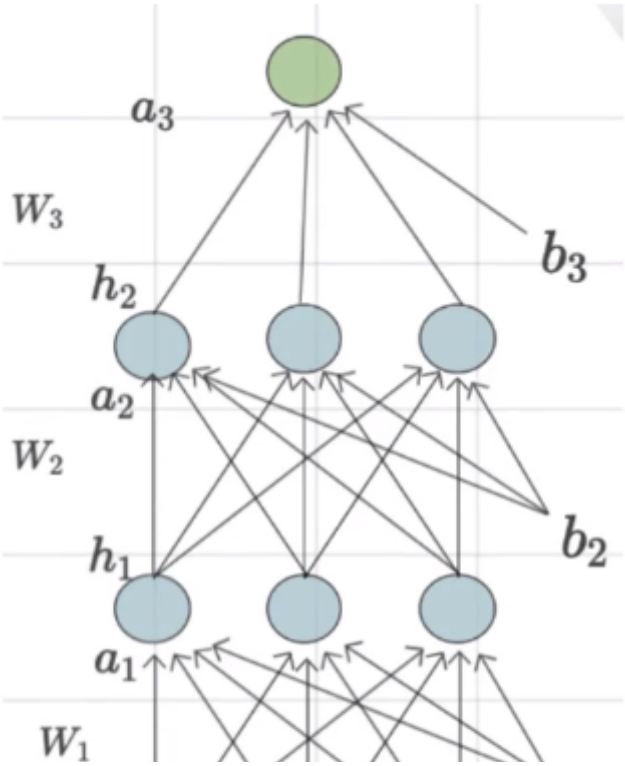
If the inputs are not normalized, then also the weighted sum of the inputs could blow up.

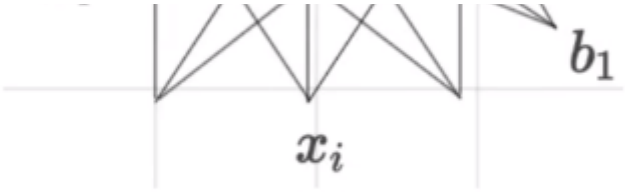


Let's see what really works in practice in terms of weights initialization.

Xavier and He initialization:

In the below network





$$a_2 = W_2 x + b_2$$
$$h_2 = \sigma(a_2)$$

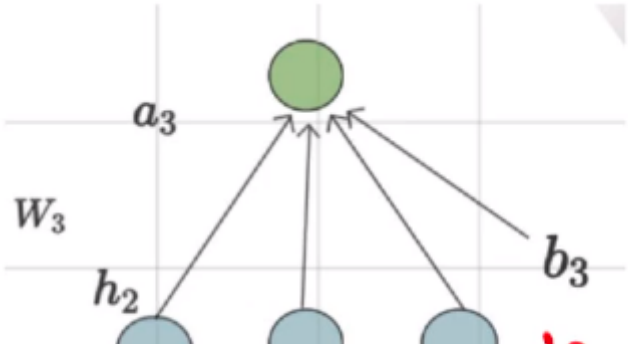
if we focus on **a2**, it would be given as:

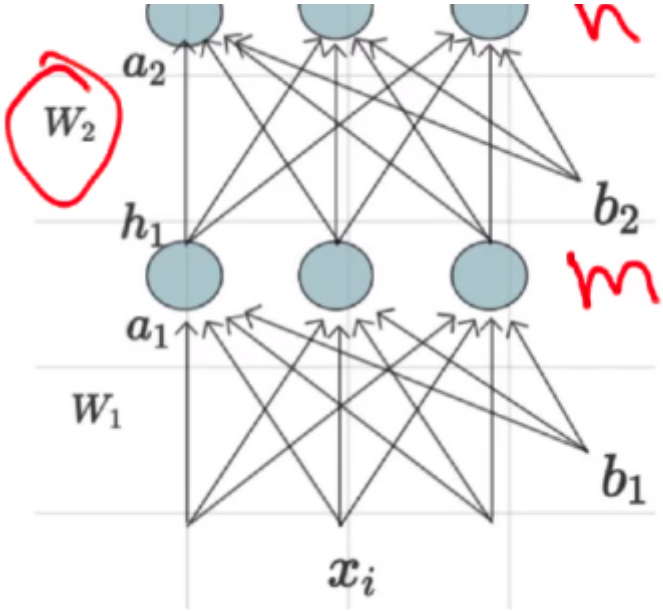
$a_{21} = w_{21} h_{11} + w_{22} h_{12} + \dots w_{2n} h_{1n}$

Now if in the input layer(which is input to the current second intermediate layer), if the number of neurons '**n**' is large

$n \to 0$

then, in that case, there is a chance that this sum will blow up. **So, it makes sense that the weights are inversely proportional to the number of neurons that we have in the previous layer.** So, if '**n**' in the previous layers is very large, the weights would take on a small value and effectively the weighted sum would be a small value. This is the intuition behind the Xavier initialization.

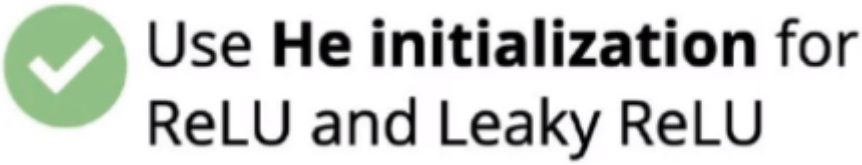




If we have **'m'** neurons in the first hidden layer and **'n'** neurons in the second hidden layer, then we need to initialize a total of **'m X n'** weights for the second intermediate layer.

And in practice, we keep the weights in the current layer inversely proportional to the number of neurons in the previous layer for the case of **logistic** and **tanh** functions.

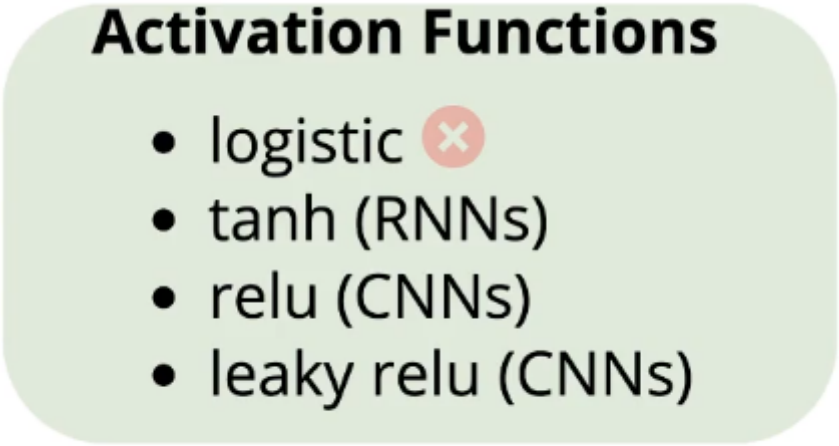
In the case of **ReLU**, we keep the weights inversely proportional to the square root of the number of neurons in the previous layer divided by 2. And the rationale for that is that in half of the region or half of the ReLU neurons are going to be in the negative region and would be dead. And this is the intuition behind '**He**' initialization.

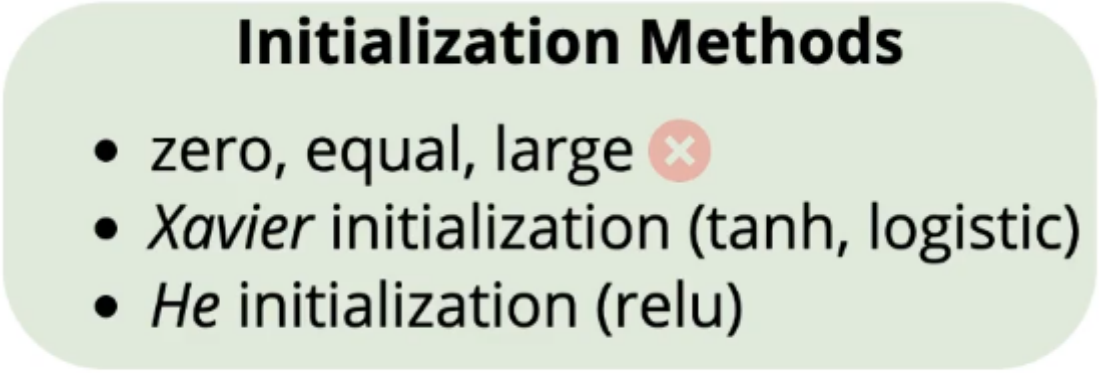


Summary:

We looked at different Activation functions and Initialization methods.

We saw that the Logistics neurons have some issues and they are not used much in practice. In the case of Recurrent Neural Network(RNN), we use **tanh** activation function and in the case of Convolutional Neural Network(CNN), we use either ReLU or leaky ReLU.





Deep Learning Activation Functions Weight Initialization Artificial Neural Network

Machine Learning